Nuclear Instruments and Methods in Physics Research A 785 (2015) 105–109

Contents lists available at ScienceDirect

Nuclear Instruments and Methods in Physics Research A

journal homepage: www.elsevier.com/locate/nima

Technical Notes

Code optimisation in a nested-sampling algorithm

S.J. Lewis [b], D.G. Ireland [b,*], W. Vanderbauwhede [a]

[a] School of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
[b] SUPA School of Physics and Astronomy, University of Glasgow, Glasgow G12 8QQ, UK

ARTICLE INFO

Article history:
Received 2 February 2015
Accepted 2 March 2015
Available online 10 March 2015

Keywords:
Monte Carlo methods
General-Purpose computation on Graphics Processing Units (GPGPU)
Data analysis

ABSTRACT

The speed-up in program running time is investigated for problems of parameter estimation with nested sampling Monte Carlo methods. The example used in this study is to extract a polarisation observable from event-by-event data from meson photoproduction reactions. Various implementations of the basic algorithm were compared, consisting of combinations of single threaded versus multi-threaded, and CPU versus GPU versions. These were implemented in OpenMP and OpenCL. For the application under study, and with the number of events as used in our work, we find that straightforward multi-threaded CPU OpenMP coding gives the best performance; for larger numbers of events, OpenCL on the CPU performs better. The study also shows that there is a "break-even" point of the number of events where the use of GPUs helps performance. GPUs are not found to be generally helpful for this problem, due to the data transfer times, which more than offset the improvement in computation time.

1. Introduction

Many data analysis tasks in nuclear and particle physics are parameter estimation problems. Values of parameters are found by comparison of a data model function with distributions of measured data. A common, and frequently satisfactory, approach is to maximise a likelihood function, and summarise the information about the parameters by taking the point of maximum likelihood and examining the behaviour in the vicinity of the maximum to estimate the uncertainty in the extracted value of the parameter. This is implemented in gradient-based searches that are the default in packages such as MINUIT [1], and is simply referred to as "fitting".

In some cases, however, such an approach is not adequate to extract all available information from experimental measurements, and a full evaluation of a likelihood function is required. This is particularly true in cases where the measured data are sparse, or where the data model has correlations among parameters that are of higher order than simply linear. By evaluating the full likelihood function, one captures all the available information, but the disadvantage can be that the calculation of likelihood may be extremely demanding.

Modern techniques of Markov Chain Monte Carlo (MCMC) calculations are designed to sample complicated, multidimensional probability density distribution functions efficiently, but whereas a gradient-based optimisation may typically require of order 100 likelihood function evaluations, a typical MCMC calculation needs perhaps of order 10^4 evaluations. Depending on the complexity of the likelihood function this could result in significant computation time.

If measurements produce event-by-event data, a likelihood of each datum can be calculated and combined to give a total likelihood. Since the same calculations need to be performed for each event, this points to the use of parallelisation to help program speed-up. The advent of general purpose graphical processor unit (GPGPU) programming suggests that an implementation of an event-by-event likelihood calculation on a GPU might be the best way forward. Indeed, it has been shown [2] that for event-by-event maximum likelihood calculations in partial wave analysis, speed-ups of two or three orders of magnitude are possible over conventional CPU running.

In this paper we examine the consequences of evaluating a likelihood function of modest complexity, where the measurements consist of event-by-event data of modest numbers. Section 2 introduces the example problem, Section 3 describes the nested sampling algorithm and outlines the various implementations in software and hardware, and Section 4 presents the results.

2. Statement of problem

The example used in this paper is based on a two-body reaction in which a linearly polarised photon interacts with a proton target, producing a pseudoscalar ($J^P = 0^-$) meson and a baryon. We wish

* Corresponding author.
E-mail address: David.Ireland@glasgow.ac.uk (D.G. Ireland).

http://dx.doi.org/10.1016/j.nima.2015.03.006

to determine the photon beam asymmetry, Σ, which is the difference divided by the sum of cross-sections for the two states of photon linear polarisation. This is achieved by measuring the distribution of mesons as a function of azimuthal angle ϕ, which is the angle between the reaction plane and the direction of the photon's linear polarisation (E-vector). We assume, for simplicity, that there is a fixed photon energy E_γ and centre-of-mass scattering angle $\theta^\star_{CM}$. In a real experiment, data would be sorted into bins with a range of E_γ and $\theta^\star_{CM}$, but the ranges would be minimised to extract the maximum physics information from the variation of observables as functions of E_γ and $\theta^\star_{CM}$. This example could be applied to any reaction in which the photon beam asymmetry is to be determined.

The cross-section as a function of the angle ϕ is given by

$$\sigma = \sigma_0(1 - P_\gamma \Sigma \cos(2\phi)) \tag{1}$$

where σ_0 is the unpolarised cross-section, P_γ is the degree of photon polarisation and Σ is the beam asymmetry we desire to extract.

We assume that it is possible to polarise the photon beam in such a way that the electric vector can be oriented either parallel ($\parallel$) or perpendicular ($\perp$) to a reference plane in the lab frame. An asymmetry between these two settings is then

$$A(\phi) = \frac{\sigma_\parallel(\phi) - \sigma_\perp(\phi)}{\sigma_\parallel(\phi) + \sigma_\perp(\phi)} = P_\gamma \Sigma \cos(2\phi). \tag{2}$$

By measuring the number of mesons as a function of ϕ in both these states, we obtain an estimator of the asymmetry:

$$\hat{A}(\phi) = \frac{N_\parallel(\phi) - N_\perp(\phi)}{N_\parallel(\phi) + N_\perp(\phi)}. \tag{3}$$

A complicating factor of potentially different numbers of incident photons in each of the two states is omitted in this example to aid clarity. The two luminosities are taken to be equal. We also assume that the degree of photon polarisation P_γ is the same for both settings, and is known accurately, so that we do not need to regard it as a nuisance parameter.

With this in mind, we see that the problem is simply a one-parameter problem, where it is knowledge of Σ that we desire to infer from the measured data. The likelihood (probability) of measuring $N_\parallel$ and $N_\perp$ events given a definite asymmetry value a is

$$\mathcal{P}(N_\parallel, N_\perp \mid a) = \frac{1}{Z}(1-a)^{N_\parallel}(1+a)^{N_\perp} \tag{4}$$

where Z is a normalising constant.

For each event, we need the meson azimuthal angle ϕ and the setting ($\parallel$ or $\perp$). For a given value of Σ, an asymmetry is calculated from Eq. (2). Eq. (4) then reduces to

$$\mathcal{P}(N_\parallel = 1, N_\perp = 0 \mid a) = \tfrac{1}{2}(1-a) \tag{5}$$

or

$$\mathcal{P}(N_\parallel = 0, N_\perp = 1 \mid a) = \tfrac{1}{2}(1+a) \tag{6}$$

depending on the setting. For M events, the total likelihood is then the product of the likelihoods of each event:

$$\mathcal{L} = \prod_{i=1}^{M} \mathcal{P}_i. \tag{7}$$

In realistic examples we expect something between 10^3 and 10^4 events, so if code can be parallelised to perform likelihood calculations on several events at once, a speed-up should be possible.

In this study to determine the best code implementation strategy for applications of this type, we have simulated the reaction with known values of Σ to generate events that we know to be free of detector peculiarities. We use a two-body phase-space generator where the azimuthal distributions of the mesons are modulated according to Eq. (1).

3. Implementation

3.1. Nested sampling

Nested sampling [3] is a form of MCMC, a Bayesian approach to inference problems. Whilst nested sampling has been applied to a specific hadron physics problem in this paper, it is a general algorithm that is applied to a wide range of problems.

The primary objective of nested sampling is to provide a sampling of a posterior probability density function, and calculate a value referred to in the literature as the *evidence*. The evidence is a quantity with which different data models can be compared, but in this application we are only interested in parameter estimation, so it is a by-product of the calculation. A *prior* probability density in parameter space is used in conjunction with the event-by-event likelihood function (Eq. (7)) to generate the posterior.

As with other MCMC applications, nested sampling works with a population of points in parameter space. The prior probability density is sampled to give an initial population, and for each point in this sampled prior, a likelihood value is calculated. In nested sampling, the point with the lowest likelihood is recorded and overwritten with a copy of a surviving point. This new point is then altered in an exploration step within parameter space, and its likelihood is calculated. If the resulting likelihood is lower than that of the overwritten point, the new point is moved again in a further exploration step. This process continues until the likelihood of the new point is found to be greater than that of the overwritten point. The algorithm then finds the next point with the lowest likelihood and the process is repeated until a given termination condition is met [4]. The general idea is that the current population will migrate to the regions of greatest probability. For consistency, the termination condition used in this work was a set number of iterations.

3.2. Data parallelisation

Since the clock speed of CPUs has stagnated, parallel programming has become the focus of computing performance development. The use of multicore processors and General-Purpose Graphics Processing Units (GPGPUs) has become mainstream in everything from scientific computing and state-of-the-art gaming technology to standard desktop computers and laptops. There is now a sustainable path to improving computing technologies for the foreseeable future. Although the spotlight is currently on GPGPU computing, it must be remembered that all programs and algorithms will include some amount of sequential code, even if it exists solely to execute kernel functions or perform or some standard initialisations. In most cases, these serial sections of a program create bottlenecks that no amount of parallelisation can avoid. For this reason, heterogeneous platforms – i.e. the combination of highly optimised CPU cores with the massively parallelisable GPU cores – have become increasingly popular. These two components must complement each other – if the CPU is outdated and obsolete, any speed-up obtained from a high-end GPU will be hidden by the slow processing at one of these bottlenecks. In order to make the most of the available hardware, both components must be taken into consideration.

Not all algorithms can be parallelised; recursive and sequential programs, or even serial sections of code, can form bottlenecks that impede the run-time of a program. There are some cases where parallelising data over multiple threads or cores can result in a slower run-time as no speed-up is gained and time is lost

during data transfer or thread initialisations. Even in cases where an algorithm lends itself naturally to parallelism, it is crucial to understand exactly where and how it should be done in order to obtain the greatest benefit.

3.3. CPU/GPU system performance

It is important to have an idea what to expect in terms of performance of a GPU-accelerated application. The achievable speed-up from offloading an application to a GPU can be analysed as a function of the computational speed up (which in turn depends on the hardware parallelism, clock speed and memory bandwidth) and the data transfer speed. Fig. 1 shows a generic graph which can be used to assess the performance of an algorithm.

What the graph shows is the achievable speed-up as a function of the CPU compute time relative to the data transfer time, with the GPU/CPU computational speed-up as a parameter. For example, if the computation on the CPU takes 100 ms, and the data transfer 1000 ms, then there can be no speed-up, no matter how fast the GPU computes. On the other hand, if the CPU takes 1000 ms and the transfer time is 100ms, then with a GPU/CPU computational speed-up of 5× the total speed-up=1000/(100+1000/5)=3.3×. The curved lines assume overlapping of data transfers and computation ("pipelined" in the legend); the straight lines assume alternating data transfers and computations. It is clear that overlapping only makes a significant difference if the data transfer and computation times are of the same order.

To facilitate the analysis, we define the computation performance indicator as *CPI=threads× SIMD vector size ×clock speed*. This number is directly proportional to flops, but more easy to obtain from datasheets. We define "threads" as the product of the number of cores and their hyperthreading capability. SIMD refers to programs that operate on many data concurrently: "Single Instruction stream Multiple Data". "SIMD vector size" is defined as the number of single-precision floating-point operations that can be performed in parallel, so on a CPU this is e.g. the width of the SSE or AVX vector instructions; on a GPU it would be the number of processing elements per compute unit. In Section 3.6 we will use the CPI to assess the expected performance of the hardware platform used in this work.

We implemented the parallel parts of the code in three different ways: OpenCL on CPU, OpenCL on GPU and OpenMP (CPU only), which we now describe briefly.

3.4. OpenCL integration

OpenCL [5] was developed by the Khronos Group in 2008 as an open standard for parallel programming of heterogeneous systems. It provides an API for control and data transfer between the host and device (typically the host CPU and a GPU) and a language for kernel development. Contrary to proprietary solutions such as Nvidia's CUDA and Microsoft's DirectX, OpenCL is open and cross-platform, so that it can be deployed on different operating systems (Linux, OS X, Windows) and hardware architectures (multicore CPUs, GPUs, FPGAs). The OpenCL API is defined for C and C++. In practice, the API is quite fine grained and verbose and requires a lot of boiler plate code to be written. Consequently, it is not straightforward to integrate OpenCL in existing codes, especially for non-computing scientists. To facilitate the integration of the OpenCL code into the existing code base, we developed the OclWrapper library [6], which supports C, C++ and Fortran-95. The library wraps the OpenCL platform, context and command queue into a single object, with a much smaller number of calls required to run an OpenCL computation. As it is a thin wrapper, the additional abstraction comes at no cost in terms of features: the OpenCL API is completely accessible.

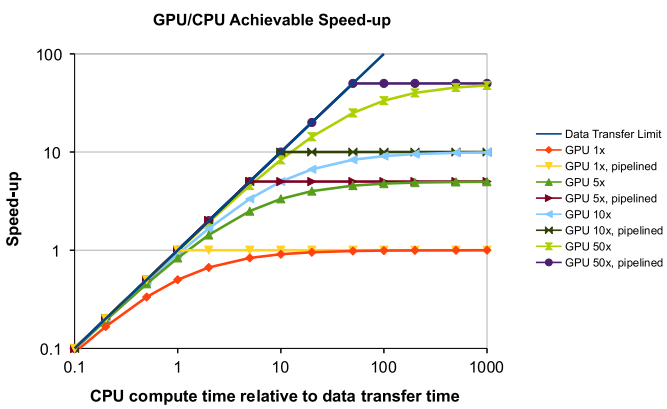

Fig. 1. Achievable speed-up from offloading work to the GPU.

3.5. OpenCL versus OpenMP

We also implemented the code in OpenMP [7], an API for parallel programming on shared-memory multicore platforms. The OpenMP API consists of a set of preprocessor directives (pragmas) and function calls.

The differences between coding in OpenMP and OpenCL are quite small, and fall into three categories: first, the OpenMP code is object-oriented C++ code, so some of the variables are class attributes and not subroutine arguments. In OpenCL all input and output variables must be kernel arguments in the `__global` memory space and the subroutine must be `__kernel void`, so the kernel signature has more arguments. Second, OpenMP requires pragmas (`pragma omp ...`) to control the parallelism and indicate which variables are shared. Finally, where the OpenMP code is identical to the original code except for the pragmas, the OpenCL kernel code consists only of the loop body.

In the OpenMP code, we compute the partial sums of log likelihoods per OpenMP thread and aggregate them. In our OpenCL code, we use a map-reduce pattern: every thread computes a partial sum, and these are reduced to the final sum. To ensure that the computation of all partial results in each thread was finished, we insert the barrier() call. Finally, we return the result for each compute unit via the array `LogL[group_id]`. The sum over all compute units is performed on the host.

3.6. Hardware platform

The host CPU used in this work is an Intel i7-2700K running at 3.5 GHz. It is a quad-core CPU with hyperthreading. The maximum memory bandwidth is 21 GB/s. This processor has 256-bit AVX SIMD, so a smart compiler will do up to 8 floating point operations in parallel. The GPU used is an NVidia Tesla S2075 GPU with 14 compute units (448 "cores") running at 1.15 GHz. The maximum memory bandwidth is 144 GB/s. The host-device connection is a 16-lane Gen2 PCIe bus. The CPIs are shown in Table 1.

Table 1
Specifications of hardware platforms used in this work.

Platform	Cores	Vector size	Clock speed (GHz)	CPI	Memory BW (GB/s)
Intel i7-2700K	12	8	3.5	336	21
Nvidia Tesla S2075	14	32	1.15	515.2	144

We observe that purely in terms of computation, under optimal circumstances, the Tesla GPU can be at best 1.5× faster than the Intel CPU. If the memory bandwidth were the limiting factor (as opposed to computation), the achievable speed-up for the application running on the GPU would be 6.8×. The total achievable speed-up is limited by the data transfer rate between host memory and GPU memory, and the overhead for control of the GPU. In Section 4.3 we present the detailed discussion of the cost of data transfer and computation. However, as our application is not memory bandwidth limited, the CPI shows that even ignoring the transfer time, the speed-up from the GPU will be very modest, and that taking into account the transfer time, we expect the GPU to perform less well than the CPU.

4. Results and discussion

In our studies, we carried out a variety of computational runs, with the objective of extracting the observable Σ, and each of which had a configuration based on the following inputs: the number of input simulated reaction events (10^3, 10^4, and 10^5); the number of threads (2–64); the number of nested sampling iterations (from 10 K to 150 K). As these calculations are based on random numbers, 20 runs were performed for each configuration to verify that variation was negligible to compute averages.

4.1. Optimal number of threads

The first tests were performed to determine the optimal number of threads using each data set on the three implementations that allowed for multithreading (OpenCL-CPU, OpenCL-GPU and OpenMP). The number of threads over which to parallelise the data was varied, and the program runtime was measured.

Figs. 2–4 illustrate that the optimal number of threads is dependent on the size of the dataset, and differs for various implementations. The large variation in the OpenMP performance is due to the fact that the OpenMP performance improves for growing numbers of threads as long as there are fewer OpenMP threads than physical threads. Once the number of OpenMP threads exceeds the number of physical threads (eight in this case), the performance deteriorates strongly. This effect is also observable for OpenCL on the CPU, but much less pronounced, because OpenCL schedules the threads so that they do not compete with one another.

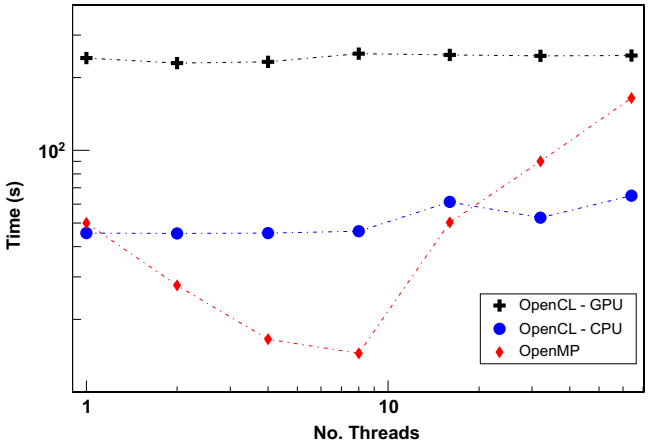

Fig. 2. Thread test results from dataset with 1000 events. Points represent the average of 20 calculations and standard deviations are smaller than the size of the plotting symbols. This is the case for Figs. 3 and 4 as well.

4.2. Optimal performance

Fig. 5 shows the performance for the optimal number of threads for each implementation, as a function of the number of simulated reaction events. We observed no dependency on the number of nested sampling iterations (5×10^4 for the results in this figure). The most interesting observation is the difference in

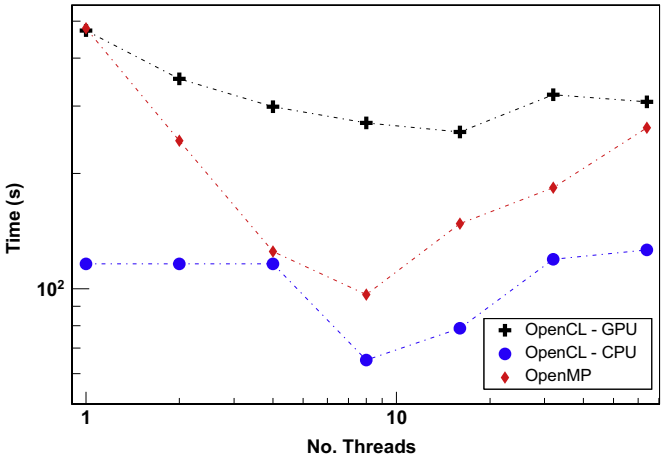

Fig. 3. Thread test results from dataset with 10 000 events.

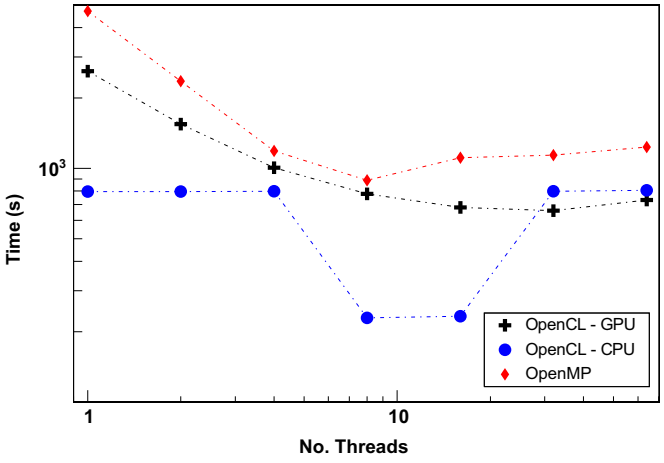

Fig. 4. Thread test results from dataset with 100 000 events.

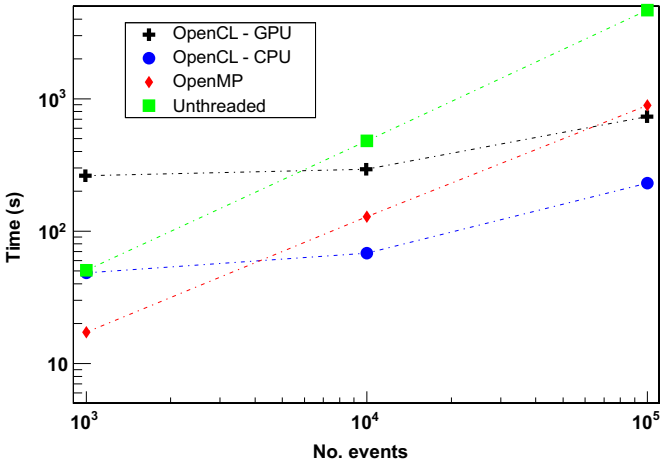

Fig. 5. Performance for optimal number of threads.

Table 2
Data transfer results per iteration for OpenCL on the GPU.

Events	Data transfer time (ms)	Total time (ms)	% Data transfer
1000	0.227 ± 0.008	0.277 ± 0.0023	82
5000	0.231 ± 0.010	0.328 ± 0.0027	70
10 000	0.232 ± 0.004	0.3997 ± 0.0041	58
100 000	0.229 ± 0.005	1.560 ± 0.0010	15

Table 3
Data transfer results per iteration for OpenCL on the CPU.

Events	Data transfer time (ms)	Total time (ms)	% Data transfer
1000	0.021 ± 0.0007	0.0457 ± 0.001	46
5000	0.021 ± 0.0022	0.0776 ± 0.001	27
10 000	0.021 ± 0.0007	0.114 ± 0.0006	18
100 000	0.021 ± 0.0011	0.790 ± 0.0074	3

behaviour between the OpenCL and OpenMP parallel code: the execution time of the OpenMP code (for 8 threads) is proportional to number of events; however, the OpenCL code performs significantly better for large numbers of events. Interestingly, the OpenCL code results in a larger speed-up than suggested by the number of hardware threads on the CPU, which indicates that the Intel OpenCL compiler also uses the vector support of the CPU. As a result of this different behaviour, there is no single optimal implementation.

For large numbers of reaction events (10^5), the best choice is OpenCL on the CPU, with a speed-up of up to $22\times$. For small numbers of events (10^3), the best choice is OpenMP, where the achievable speed-up is smaller, up to $4\times$. For intermediate numbers (10^4), the best speed-up was $8\times$ using OpenCL on the CPU. The most likely number of reaction events for applying this method to real data is 10^3–10^4. The best GPU speed-up was $7\times$ for 10^5 events. This is mainly a result of the data transfer time, which more than offsets the improvement in computation time.

4.3. Impact of data transfer time

The amount of time spent merely transferring data to a given compute device using OpenCL was measured by running the program with an empty likelihood kernel function (i.e. the likelihood function does not calculate anything, but merely returns with a number, and the nested sampling run is carried out for a fixed number of iterations). The results are given in Tables 2 and 3.

The data transfer time is linear with the data set size but has a significant constant offset. This offset is the time taken to control the GPU, i.e. not the actual transfer time. The corresponding time for OpenCL on the CPU is about $10\times$ smaller. The number of reaction events used in this work are typical for the problem area discussed in this paper, but are fairly small in absolute size (e.g. PWA maximum likelihood calculations may involve 10^5–10^6 events).

These results illustrate that a significant portion of the runtime for the GPU is spent handling the control of the device. In algorithms such as nested sampling, where the kernel is invoked many times for relatively small calculations, data parallelism on the GPU is not necessarily the most effective solution. Parallelising data and running on the CPU provides a much more noticeable speed-up, partly due to this overhead. This is in line with our CPI-based analysis.

5. Conclusion

We have studied how to optimise the performance of nested sampling calculations, where parameters are to be estimated from event-by-event likelihoods. Our work demonstrates that OpenCL can be used successfully to accelerate the program. Thanks to our novel OclWrapper library, the OpenCL integration requires only a small additional effort compared to OpenMP. Our work also shows that the best choice of implementation and hardware platform depends very much on the number of events in a data set, when using event-by-event likelihood calculations. In any case, for this particular algorithm, the amount of computations is too small to outweigh the cost of the data transfer to the GPU, so a multicore CPU is a better choice. For large numbers of events (10^5), we achieved a speed-up of up to 22 times. For the number of events likely to be measured in a real experiment ($\sim 10^3$–10^4), however, the OpenMP implementation is the best option.

In future work we want to investigate if we can convert more portions of the nested sampling algorithm into OpenCL kernels, in order to reduce the data transfer between host and device. Perhaps the most important message from our study is that the optimum choice of hardware/software implementation is very problem-specific.

Acknowledgements

The authors are grateful for the support of the United Kingdom's Science and Technology Facilities Council (STFC) consolidated Grant ST/J000175/1.

References

[1] F. James, M. Roos, Computer Physics Communications 10 (1975) 343. http://dx.doi.org/10.1016/0010-4655(75)90039-9.
[2] N. Berger, Journal of Physics: Conference Series 331 (3) (2011) 032005, URL: ⟨http://stacks.iop.org/1742-6596/331/i=3/a=032005⟩.
[3] J. Skilling, Bayesian Analysis 1 (4) (2006) 833.
[4] D. Sivia, J. Skilling, Data Analysis: A Bayesian Tutorial, Oxford Science Publications, OUP, Oxford, 2006, URL: ⟨http://books.google.de/books?id=zN-yliq6eZ4C⟩.
[5] A. Munshi, et al., The OpenCL Specification, Khronos OpenCL Working Group 1, 2009, pp. l1–15.
[6] W. Vanderbauwhede, OpenCLIntegration URL: ⟨https://github.com/wimvanderbauwhede/OpenCLIntegration⟩ 2013.
[7] L. Dagum, R. Menon, IEEE Computational Science & Engineering 5 (1) (1998) 46.